K.N. Toosi University

Aidin Sahneh

Student ID: 40120243

Course: Digital systems 2 lab

Instructor: Dr. Aslani

Final Project

Contents

1 Schematic Design and Implementation

The foundation of this project is a comprehensive hardware simulation designed in Proteus, which emulates the real-world behavior of the microcontroller and its connected peripherals. The schematic, shown in Figure 1, serves as the platform for developing and testing the embedded C code before deployment on physical hardware.

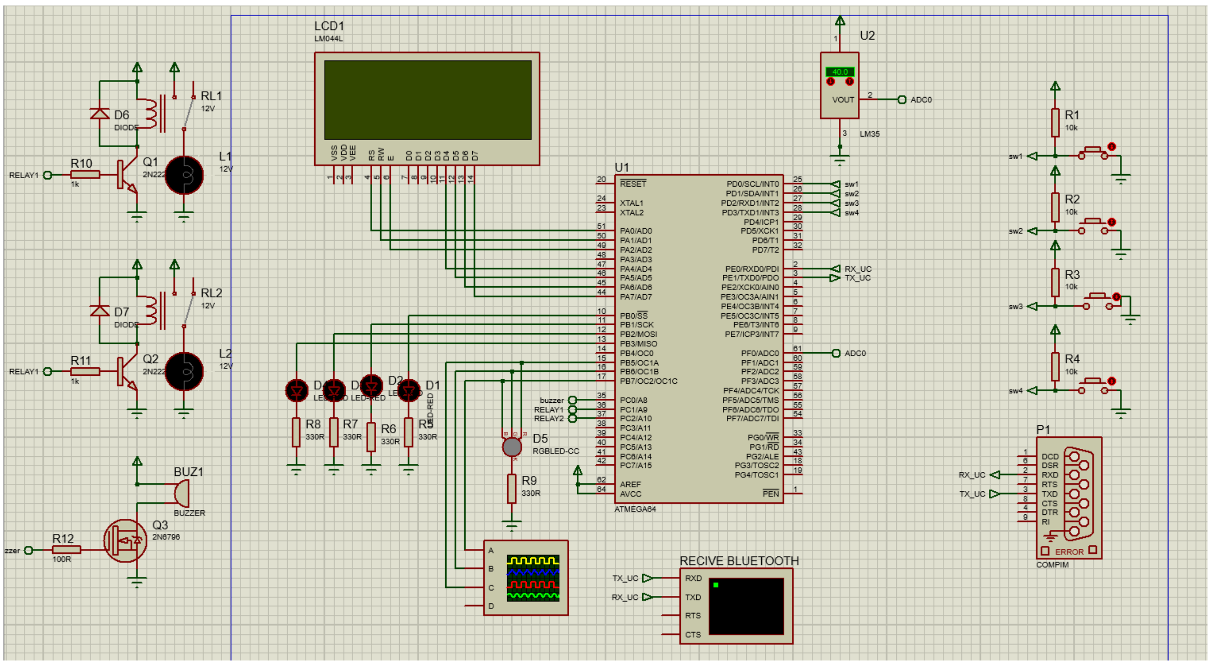


Figure 1: The complete project schematic designed in Proteus.

The circuit is centered around the **ATmega64A microcontroller (U1)**, which acts as the brain of the entire system. It is responsible for executing the control logic, reading sensor data, processing user inputs, and managing all output devices. The construction of the schematic can be broken down into its core input and output subsystems.

1.1 Input Subsystems

The input subsystems provide the necessary data and commands for the microcontroller to act upon.

- **User Push-Buttons (SW1-SW4):** Four momentary push-buttons are connected to pins PD0 through PD3. Each button is connected with a 10kΩ pull-up resistor to VCC, ensuring a stable HIGH signal when idle. When a button is pressed, the corresponding pin is pulled to GROUND, allowing the microcontroller to detect a button press as a falling edge. These buttons provide manual control over features like temperature unit conversion and LED blinking modes.
- **Temperature Sensor (LM35):** An LM35 analog temperature sensor is connected to pin PF0 (ADC0). This sensor outputs a voltage that is linearly proportional to the temperature (10mV per degree Celsius). This analog signal is fed directly into the microcontroller's Analog-to-Digital Converter (ADC) for processing.

- **Bluetooth Communication (Virtual Terminal):** To simulate the HC-05 or similar Bluetooth module, a **Virtual Terminal** component is used. It is wired for two-way UART communication. The terminal's Transmit (TXD) pin is connected to the microcontroller's Receive pin (PE0/RXD0), and the terminal's Receive (RXD) pin is connected to the microcontroller's Transmit pin (PE1/TXD0). This crossover connection is essential for sending commands from the user to the microcontroller and receiving status messages back.

1.2 Output Subsystems

The output subsystems are controlled by the microcontroller to interact with the user and the environment.

- **LCD Display (LM044L):** A 20x4 character LCD is connected to PORTA in 8-bit mode. This display serves as the primary user interface, providing real-time feedback on the system's status, including temperature, device states (LEDs, relays, buzzer), and any text received via Bluetooth.

- **Indicator LEDs (D1-D4):** Four standard LEDs are connected to pins PB0 through PB3. Each LED is paired with a 330Ω current-limiting resistor to protect both the LED and the microcontroller's output pins from excessive current.

- **RGB LED (D5):** A common-cathode RGB LED is used for color output. Its three pins are connected to the microcontroller's hardware PWM (Pulse Width Modulation) outputs: PB5 (OC1A) for Red, PB6 (OC1B) for Green, and PD7 (OC2) for Blue. By varying the duty cycle of the PWM signals, the microcontroller can control the intensity of each color element to produce a wide spectrum of colors.

- **Relays and Driver Circuits (RL1, RL2):** Two 12V relays are connected to pins PC1 and PC2. Since the microcontroller cannot directly drive the high-current relay coils, a driver circuit consisting of a **2N2222 NPN transistor** is used for each. A $1k\Omega$ base resistor limits the current from the microcontroller pin. A crucial **flyback diode** is placed in parallel with each relay coil to safely dissipate the back EMF voltage spike that occurs when the coil is de-energized, thereby protecting the transistor from damage.

- **Buzzer (BUZ1):** An active buzzer is connected to pin PC0 through a similar NPN transistor driver circuit. This allows the microcontroller to switch the buzzer on and off to provide audible alerts.

2 Project Implementation Steps

2.1 Step 1: System Status Display on LCD

The primary objective of this step is to create a comprehensive user interface on the 20x4 character LCD. This display provides the user with real-time feedback on all critical system parameters, including the ambient temperature from the LM35 sensor and the current on/off status of all connected output devices (LEDs, relays, and the buzzer).

2.1.1 Temperature Sensing and Conversion

The process begins by reading the analog voltage from the LM35 temperature sensor. This is handled by a dedicated function, `read_adc()`, which is called within the main program loop. This function configures the ADC, initiates a conversion on channel 0, and returns the 10-bit digital representation of the sensor's output voltage.

```
unsigned int read_adc(unsigned char adc_input)
{
ADMUX=adc_input | (1<<REFS0); // Select channel and AVCC reference
delay_us(10);
ADCSRA|=(1<<ADSC);            // Start conversion
while ((ADCSRA & (1<<ADIF))==0); // Wait for conversion to complete
ADCSRA|=(1<<ADIF);            // Clear interrupt flag
return ADCW;                  // Return 10-bit result
}
```

Listing 1: Function to read a value from the ADC.

Once the raw digital value is obtained, it must be converted into a meaningful temperature in degrees Celsius. This is achieved using the following formula, which accounts for the ADC's 5V reference voltage and the LM35's linear scale factor of 10mV/°C. The result is stored in a global variable named `temperature`.

```
temperature = (read_adc(0) * 500.0) / 1024.0;
```

Listing 2: Conversion of raw ADC value to degrees Celsius.

2.1.2 State Management and Display Logic

To accurately display the status of each output device, the software maintains a set of global `bit` variables (e.g., `led1_state`, `relay1_state`). These variables act as a "single source of truth" for the system's state and are updated in real-time whenever a command is received.

The core of the display logic resides inside the main `while(1)` loop. During each iteration, the LCD is cleared and then repopulated line by line. The `sprintf()` function is used to format the output strings, dynamically embedding the current temperature and the on/off status of each device. A ternary operator (`condition ? "ON " : "OFF"`) provides a concise way to select the correct status string based on the corresponding state variable.

To prevent data corruption on the display, which could be caused by an interrupt firing in the middle of a multi-line update, the entire LCD writing block is protected by disabling interrupts with `#asm("cli")` before it begins and re-enabling them with `#asm("sei")` immediately after it concludes.

```
// --- Update LCD Display ---
#asm("cli") // Disable interrupts to prevent screen tearing
```

```c
lcd_clear();

// Line 0: Temperature and Buzzer Status
sprintf(display_buffer, "Temp:%.0f C   Bzr:%s", temperature, buzzer_state ?
    "ON " : "OFF");
lcd_gotoxy(0, 0); lcd_puts(display_buffer);

// Line 1: LED 1, 2 and Relay 1 Status
sprintf(display_buffer, "L1:%s L2:%s R1:%s", led1_state ? "ON ":"OFF",
    led2_state ? "ON ":"OFF", relay1_state ? "ON ":"OFF");
lcd_gotoxy(0, 1); lcd_puts(display_buffer);

// Line 2: LED 3, 4 and Relay 2 Status
sprintf(display_buffer, "L3:%s L4:%s R2:%s", led3_state ? "ON ":"OFF",
    led4_state ? "ON ":"OFF", relay2_state ? "ON ":"OFF");
lcd_gotoxy(0, 2); lcd_puts(display_buffer);

// Line 3: Text received via Bluetooth
lcd_gotoxy(0, 3); lcd_puts(received_text);

#asm("sei") // Re-enable interrupts
```
Listing 3: Code for formatting and writing data to the LCD.

2.2 Step 2: Individual Device Control via Bluetooth

This step implements remote control of all individual output devices (four LEDs, two relays, and one buzzer) from the ”Switches” page of the mobile application. The application is configured to send a unique single character for the ’ON’ and ’OFF’ state of each switch.

The software uses an interrupt-driven approach to handle incoming serial data. When a character is received via Bluetooth, the `usart0_rx_isr()` interrupt service routine places the character into a circular buffer. The main loop continuously checks if there is data in this buffer. If so, it retrieves the character and processes it through a command handler.

A `switch` statement is used to efficiently parse the received command. Each `case` corresponds to a character sent by the app. For example, receiving an uppercase ’A’ turns on LED 1, while a lowercase ’a’ turns it off. When a command is matched, the code performs two actions:

1. It directly manipulates the corresponding port pin to change the physical state of the device (e.g., `PORTB.0=1;`).
2. It updates the associated global state variable (e.g., `led1_state=1;`) to ensure the LCD display remains synchronized with the hardware’s actual state.

An important feature of this logic is that the LED control commands are nested inside an `if (blinking_mode_active == 0)` block. This prevents the user from turning individual LEDs on or off while the global blinking mode is active, avoiding conflicting states.

```c
// --- LED Control (only if not in blinking mode) ---
if (blinking_mode_active == 0) {
switch (command) {
case 'A': PORTB.0=1; led1_state=1; break;
case 'a': PORTB.0=0; led1_state=0; break;
// ... cases for B/b, C/c, D/d ...
}
}
```

```c

// --- Relay & Buzzer Control ---
switch(command) {
case 'E': PORTC.1=1; relay1_state=1; break;
case 'e': PORTC.1=0; relay1_state=0; break;
// ... cases for F/f, G/g ...
}
```

Listing 4: Command handler for individual device control.

2.3 Step 3: Simultaneous Relay Control

To provide a more convenient user experience, this step implements the simultaneous control of both relays using the ”LED/Lamp” page in the mobile application. This page features a single large button that sends one character for ’ON’ and another for ’OFF’.

The implementation leverages the same command parsing logic described in the previous step. Specific characters, ’L’ (uppercase) for ON and ’l’ (lowercase) for OFF, have been designated for this function. When the command handler receives one of these characters, it bypasses the main `switch` statement and is handled by a dedicated `if/else if` block.

Upon receiving an ’L’, the code sets the port pins for both Relay 1 (PC1) and Relay 2 (PC2) to HIGH and updates their respective state variables. Conversely, upon receiving an ’l’, it sets both pins to LOW and updates the state variables accordingly. This provides a simple and effective way to control multiple devices with a single command.

```c
// Single-character device control
} else {
/************************************************************
* REQUIREMENT 3-3: Control both relays from LED/Lamp page.
* We assign 'L' for ON and 'l' for OFF to control both relays.
************************************************************/
if (command == 'L') {
PORTC.1=1; relay1_state=1;
PORTC.2=1; relay2_state=1;
} else if (command == 'l') {
PORTC.1=0; relay1_state=0;
PORTC.2=0; relay2_state=0;
}
// ... rest of switch statements for individual control ...
}
```

Listing 5: Code for simultaneous relay control.

2.4 Step 4: Temperature Unit Conversion

This feature provides the user with the flexibility to view the ambient temperature in either Celsius or Fahrenheit. The functionality is controlled by the on-board push-button, SW1.

The software continuously monitors the state of the pin connected to SW1 (PIND.0). To ensure that a single, prolonged press is not registered as multiple events, an edge-detection algorithm is implemented. The code checks for a transition from a HIGH state (button not pressed) to a LOW state (button pressed). This is achieved by comparing the button’s current state to its state from the previous loop iteration, which is stored in the `last_sw1_state` variable.

Upon detecting a valid press, a global boolean flag, `is_fahrenheit`, is inverted (toggled). A small 50ms delay is included for software debouncing, preventing false triggers from mechanical contact bounce.

```
// --- Handle On-board Push-Buttons ---
current_sw1_state = PIND.0; // Temp Unit Toggle
if ((last_sw1_state == 1) && (current_sw1_state == 0)) {
is_fahrenheit = !is_fahrenheit;
delay_ms(50); // Debounce delay
}
last_sw1_state = current_sw1_state;
```

Listing 6: Code for handling the SW1 button press.

This `is_fahrenheit` flag is then used by the LCD update logic. An `if/else` block checks the flag's value and formats the temperature string accordingly, either displaying the direct Celsius value or calculating and displaying the Fahrenheit equivalent using the standard conversion formula $(T_C \times \frac{9}{5}) + 32$.

2.5 Step 5 & 6: LED Blinking Mode and Speed Control

This functionality allows the user to switch the four indicator LEDs into a synchronized blinking mode and adjust the blinking frequency using the on-board push-buttons SW2, SW3, and SW4.

2.5.1 Activating Blinking Mode (SW2)

Push-button SW2 acts as a toggle for the blinking mode. Using the same edge-detection and debouncing technique as SW1, the code monitors pin PIND.1. When a press is detected, the `blinking_mode_active` boolean flag is toggled.

A crucial part of this logic is that when the blinking mode is deactivated (`blinking_mode_active` becomes 0), the code immediately sets all four LED pins to LOW. This ensures the LEDs are turned off and the system returns to a clean, predictable state, rather than leaving the LEDs frozen in whatever state they were in when the mode was disabled.

```
current_sw2_state = PIND.1; // Blinking Mode Toggle
if ((last_sw2_state == 1) && (current_sw2_state == 0)) {
blinking_mode_active = !blinking_mode_active;
// If blinking is turned off, ensure all LEDs are also turned off.
if (blinking_mode_active == 0) {
PORTB.0=0; PORTB.1=0; PORTB.2=0; PORTB.3=0;
}
delay_ms(50);
}
last_sw2_state = current_sw2_state;
```

Listing 7: Toggling the LED blinking mode with SW2.

2.5.2 Blinking Logic and Speed Control (SW3 & SW4)

The blinking effect itself is implemented within the Timer0 overflow interrupt service routine, `timer0_ovf_isr`. This ISR executes approximately every 32.768ms. Inside the ISR, the code first checks if `blinking_mode_active` is true. If it is, a counter (`timer_counter`) is incremented. When this counter reaches a threshold value defined by `blinking_speed_ticks`, the state of all four LED port pins is inverted, causing them to blink. The counter is then reset.

```
// --- Timer 0 Interrupt (approx every 32.768ms) ---
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
// ... data sending logic ...

// Blinking LED Logic
if (blinking_mode_active)
{
timer_counter++;
if (timer_counter >= blinking_speed_ticks)
{
PORTB.0=!PORTB.0; PORTB.1=!PORTB.1;
PORTB.2=!PORTB.2; PORTB.3=!PORTB.3;
timer_counter = 0; // Reset counter
}
}
}
```

Listing 8: Timer interrupt logic for blinking the LEDs.

The user can adjust the blinking speed using SW3 (to increase speed) and SW4 (to decrease speed). These buttons, connected to PIND.2 and PIND.3, use the same debounced edge-detection logic. A press on SW3 decrements the `blinking_speed_ticks` variable, while a press on SW4 increments it. By changing this threshold, the user directly controls the period of the blinking, as a smaller tick count results in a faster blink and a larger count results in a slower blink. The value is constrained between 2 and 30 to ensure the period remains within the project's specified range of approximately 100ms to 2s.

```
current_sw3_state = PIND.2; // Blinking Faster
if ((last_sw3_state == 1) && (current_sw3_state == 0)) {
if (blinking_speed_ticks > 2) blinking_speed_ticks--;
delay_ms(50);
}
last_sw3_state = current_sw3_state;

current_sw4_state = PIND.3; // Blinking Slower
if ((last_sw4_state == 1) && (current_sw4_state == 0)) {
if (blinking_speed_ticks < 30) blinking_speed_ticks++;
delay_ms(50);
}
last_sw4_state = current_sw4_state;
```

Listing 9: Adjusting the blinking speed with SW3 and SW4.

2.6 Step 7: Two-Way Terminal Communication

A key requirement of the project is to establish a two-way communication link between the microcontroller and the user via the "Terminal" page of the mobile application. This involves two distinct functionalities: the microcontroller periodically sending status updates to the user, and the user sending arbitrary text messages to be displayed on the device's LCD.

2.6.1 Automatic Status Transmission

To provide the user with real-time remote monitoring, the system is programmed to automatically transmit a status string every 500 milliseconds. This process is managed efficiently using the Timer0 overflow interrupt.

Inside the timer0_ovf_isr, a counter variable (ms_counter) is incremented on each interrupt cycle (every ~32.768ms). When this counter reaches a value of 15, corresponding to approximately 500ms, it sets a global flag called send_data_flag and resets itself. Using a flag-based system is critical for performance, as it avoids executing the slow sprintf() and puts() functions directly within the time-sensitive interrupt routine.

```
// --- Timer 0 Interrupt (approx every 32.768ms) ---
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
// Automatic Data Sending Logic (Step 7)
ms_counter++;
if (ms_counter >= 15) // ~500ms
{
send_data_flag = 1;
ms_counter = 0;
}
// ... blinking logic ...
}
```

Listing 10: Timer interrupt logic for flagging a data transmission.

The main while(1) loop continuously polls this flag. When it detects that send_data_flag has been set, it formats a status string containing the current temperature (in the user-selected unit) and the state of both relays. This string is then sent out via the USART using the puts() function. Finally, the flag is cleared, and the system waits for the next 500ms interval.

```
// --- Send Automatic Data via Bluetooth ---
if (send_data_flag)
{
char tx_buffer[40];
sprintf(tx_buffer, "Temp:%.0f%c, R1:%s, R2:%s\r\n",
is_fahrenheit ? (temperature * 9.0 / 5.0) + 32.0 : temperature,
is_fahrenheit ? 'F' : 'C',
relay1_state ? "ON" : "OFF",
relay2_state ? "ON" : "OFF");
puts(tx_buffer);
send_data_flag = 0; // Clear the flag
}
```

Listing 11: Sending the status string from the main loop.

2.6.2 Receiving and Displaying Text

The system is also capable of receiving and displaying a text message of up to 20 characters from the user. The command parsing logic in the main loop is configured to handle incoming text.

As characters are received, they are appended one by one to a character array called received_text. The code checks to ensure the characters are printable (ASCII values 32-126) and that the buffer length does not exceed 20 characters. The reception of a carriage return character ('
r'), which the mobile app sends when the user presses 'send', signifies the end of the message. Upon receiving this character, a null terminator ('
0') is appended to the string, and the buffer index is reset, making the system ready to receive the next message.

This received_text string is then displayed on the fourth and final line of the LCD during every update cycle, providing a persistent display of the last message sent by the user.

```
// Terminal text parsing
} else if (command == '\r') {
received_text[received_text_index] = '\0'; // Null-terminate the string
received_text_index = 0; // Reset for next message
} else if (command >= 32 && command <= 126 && received_text_index < 20) {
received_text[received_text_index++] = command; // Append character
// Single-character device control
} else {
// ... device control logic ...
}
```

Listing 12: Logic for receiving and buffering incoming text.

2.7 Step 8: RGB LED Control with PWM

The final feature of the project is the ability to control a common-cathode RGB LED from the ”RGB Picker” page of the mobile application. This is accomplished by using the ATmega64A's built-in hardware timers to generate three independent Pulse Width Modulation (PWM) signals, one for each of the primary colors (Red, Green, Blue).

2.7.1 Hardware Timer and PWM Initialization

To generate the necessary signals, three separate 8-bit Fast PWM channels are configured during the microcontroller's initialization sequence.

- **Timer/Counter1** is configured to provide two PWM channels: Channel A (OC1A on pin PB5 for Red) and Channel B (OC1B on pin PB6 for Green).
- **Timer/Counter2** is configured to provide one PWM channel (OC2 on pin PD7 for Blue).

The configuration is set in the `TCCR1A`, `TCCR1B`, and `TCCR2` registers. The duty cycle of each channel, which controls the brightness of the corresponding color, is determined by the value written to its Output Compare Register (`OCR1AL`, `OCR1BL`, and `OCR2`).

```
// In main() function
TCCR1A=0xA1; TCCR1B=0x0B; // Timer1, 8-bit Fast PWM on OC1A & OC1B
TCCR2=0x6B; TCNT2=0x00;   // Timer2, 8-bit Fast PWM on OC2
```

Listing 13: Initialization of Timer1 and Timer2 for 8-bit Fast PWM.

2.7.2 Bluetooth Command Parsing and Color Conversion

The mobile application's RGB color picker transmits the selected color as a 7-character string, beginning with a hash symbol followed by a 6-digit hexadecimal code representing the Red, Green, and Blue values (e.g., `#RRGGBB`).

The software uses a simple state machine to parse this incoming data stream.

1. The reception of a `'#'` character signals the start of a new color command and initializes a buffer index.
2. The next six incoming characters are stored sequentially in a character array, `rgb_buffer`.
3. Once all six characters have been received, a helper function, `HexToRGB()`, is called.

The HexToRGB() function processes the 6-character string, converting each two-character hex pair into a single 8-bit integer value (0-255). This provides the three distinct intensity values for the red, green, and blue channels.

```c
unsigned char hex_char_to_int(char c){
if (c >= '0' && c <= '9') return c - '0';
if (c >= 'A' && c <= 'F') return c - 'A' + 10;
if (c >= 'a' && c <= 'f') return c - 'a' + 10;
return 0;
}

void HexToRGB(char *hex, unsigned char *r, unsigned char *g, unsigned char
    *b){
*r = (hex_char_to_int(hex[0]) * 16) + hex_char_to_int(hex[1]);
*g = (hex_char_to_int(hex[2]) * 16) + hex_char_to_int(hex[3]);
*b = (hex_char_to_int(hex[4]) * 16) + hex_char_to_int(hex[5]);
}
```

Listing 14: Helper functions for converting hex strings to RGB values.

2.7.3 Updating the LED Color

After the hex string has been converted into three 8-bit integer values, these values are written directly to the corresponding Output Compare Registers. This action immediately updates the duty cycle of each PWM signal, changing the brightness of each individual color die in the RGB LED and producing the final mixed color selected by the user.

```c
// RGB command parsing (e.g., #RRGGBB)
if (command == '#') {
rgb_buffer_index = 1;
} else if (rgb_buffer_index > 0 && rgb_buffer_index < 7) {
rgb_buffer[rgb_buffer_index - 1] = command;
rgb_buffer_index++;
if (rgb_buffer_index == 7) { // All 6 characters received
unsigned char red, green, blue;
HexToRGB(rgb_buffer, &red, &green, &blue);
// Update the PWM duty cycles
OCR1AL = red;
OCR1BL = green;
OCR2 = blue;
rgb_buffer_index = 0; // Reset for next command
}
}
```

Listing 15: Parsing the incoming string and updating PWM registers.

3 Simulation and Debugging

3.1 Virtual Terminal Configuration

In the Proteus simulation environment, a single Virtual Terminal component is used to emulate the two-way communication channel of a physical Bluetooth module. Serial communication, by its nature (specifically the UART protocol), is full-duplex, meaning it can transmit and receive data simultaneously. To accurately model this, one terminal is configured for both input and output.

This is achieved through a crucial crossover connection:

- The terminal's Transmit pin (**TXD**) is connected to the microcontroller's Receive pin (**RXD0**). This allows characters typed in the terminal to be sent to the microcontroller.
- The terminal's Receive pin (**RXD**) is connected to the microcontroller's Transmit pin (**TXD0**). This allows data sent from the microcontroller to be displayed in the terminal window.

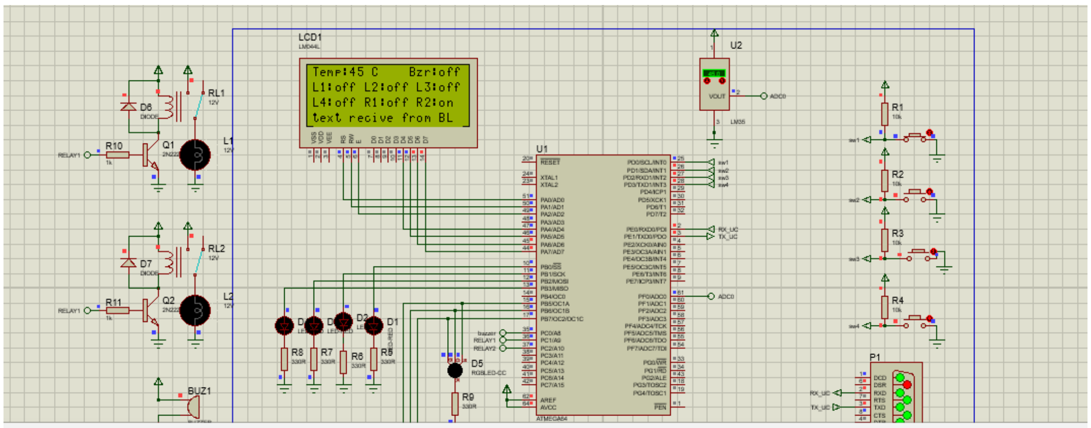
This single-terminal setup is not only more efficient but also more accurately represents how a real-world serial interface operates, where a single console is used for both sending commands and viewing responses.

4 Results and Verification

The functionality of the system was verified by running the simulation in Proteus and observing the outputs on the LCD and Virtual Terminal in response to various inputs.

4.1 Initial System State

Figure 2 shows the system's state immediately after startup. The LM35 sensor reads an ambient temperature of 45°C, which is correctly displayed on the first line of the LCD. The status of all connected devices is shown, with all LEDs and Relay 1 being off, and Relay 2 being on by default. The fourth line displays the default message, which persists until a new message is received from the user via the terminal.



```
Temp:45 C    Bzr:off
L1:off L2:off L3:off
L4:off R1:off R2:on
text recive from BL
```

Figure 2: The LCD displaying the initial temperature and device status.

4.2 Fahrenheit Conversion Test

To verify the functionality of the temperature unit conversion, the on-board push-button SW1 was pressed. As shown in Figure 3, the system correctly converts the temperature reading and updates the display. The LM35 sensor indicates a temperature of 40.0°C, and the LCD correctly displays this value converted to 104°F. This confirms that the button press logic and the conversion formula are implemented correctly.

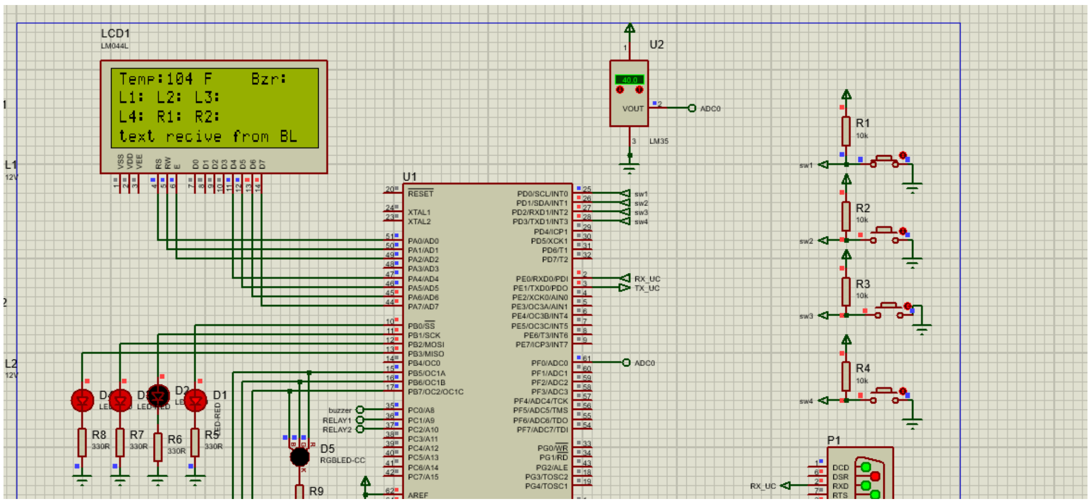
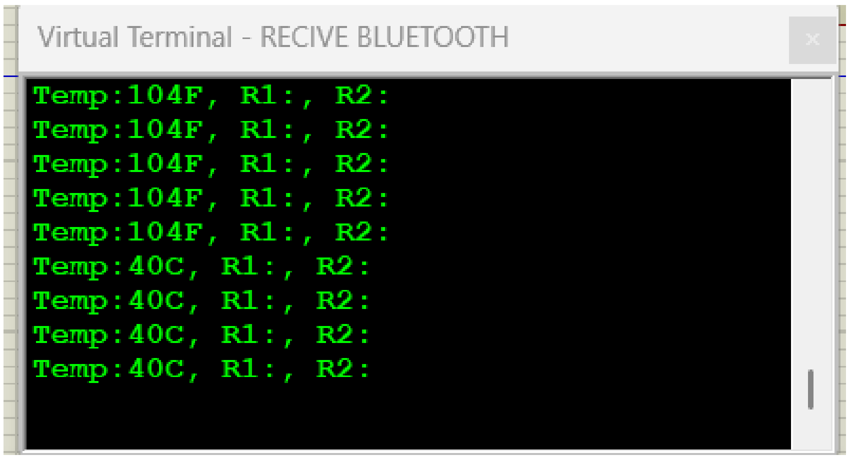


Figure 3: The LCD displaying the temperature in Fahrenheit after SW1 is pressed.

4.3 Serial Terminal Output Verification

The automatic transmission of status data was verified using the Virtual Terminal, as shown in Figure 4. The terminal displays a continuous stream of status messages, sent from the microcontroller every 500ms. The output correctly shows the temperature and the on/off status of the two relays. The image also demonstrates that when the temperature unit is toggled using SW1, the transmitted data immediately reflects this change, switching from sending ”104F” to ”40C”. This confirms the two-way communication link and the dynamic nature of the status updates.



```
Temp:104F, R1:, R2:
Temp:104F, R1:, R2:
Temp:104F, R1:, R2:
Temp:104F, R1:, R2:
Temp:104F, R1:, R2:
Temp:40C, R1:, R2:
Temp:40C, R1:, R2:
Temp:40C, R1:, R2:
Temp:40C, R1:, R2:
```

Figure 4: The Virtual Terminal showing automatic status updates, correctly reflecting the change from Fahrenheit to Celsius.